Compiling C# for Systems Programming

David Tarditi

Midori OS Brownbag Series

August 8, 2013

Overview

Midori is written almost entirely in C#

- Use C# for its developer productivity and reliability advantages
- Basis for Midori's concurrency safety and lightweight processes

Today's talk

- How to compile C# so that you can use it to write an OS
- A 35,000 foot overview of Bartok, the compiler underlying Midori
- Will schedule future talks based on researcher interest

Approaches

Compile ahead-of-time

Reduce the memory and time overheads of managed code

Top 5 features:

- Generic sharing
- Shared libraries
- Class initialization at process start-up
- Frozen objects
- Efficient linked stacks for concurrency

Will discuss shared library implementation in more depth

Highly optimizing compiler

- Same optimization capabilities as production C++ compilers.
- Extend for managed code.

Midori OS

Runs on x86, x64, ARM

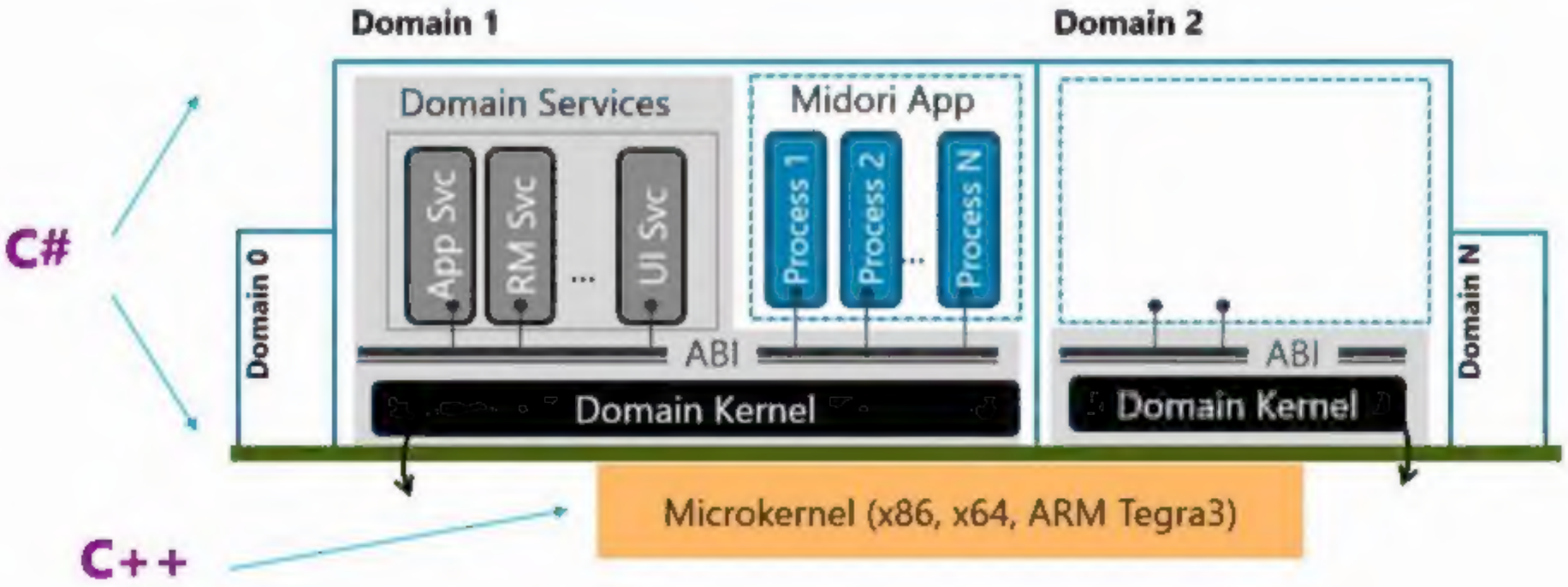
- Small, policy-free microkernel (C++) for HW interaction
- Domain kernel (C#) contains most policy: schedulers, memory, resource management

Isolation in depth

- Software isolation between processes (type-safety)
- Hardware isolation between domains for untrusted code

Type safety, capability-based security

- Most of the system
- All 3rd party code, including device drivers as processes



Case Study: SPECWeb05 (from March '13 Midori talk)

Midori can deliver performance competitive with Windows

Number of conforming connections (higher is better)

Workload	Windows	Midori	Ratio
Banking	34,800	69,000	198%
eCommerce	70,600	82,500	117%
Support (4 core)	27,300	47,000	172%
Weighted Average	**49,461**	**78,442**	**159%**

System

2x Quad Core CPU, 48GB RAM, 4x 512GB SSD, 2x 10Gb NIC + 2x 1Gb NIC
Logging and HTTPS enabled for both Windows and Midori runs per benchmark rules.
HyperThreading enabled.

Windows Settings

Same as official Windows submissions on SPEC.org, except for custom tunings that improves Windows performance such as affinity.
IPSEC and Firewall off

http://midori/Wiki Pages/Comparison of Midori and Windows SpecWeb05 Performance.aspx

Ahead-of-time compilation

Compiler creates stand-alone executables:

Instantiate all generic types at compile time

For core system, drop C# features that rely on a JIT

Dynamic class loading, reflection, unbounded polymorphic recursion, generic virtual methods

Still use a JIT for dynamic scripting languages

Ahead-of-time compilation

Advantages

- Faster machine code – time/memory for optimization not highly constrained
- Avoids memory and runtime cost of JIT
- Reuse existing OS approaches (for example, debugging, crash dumps)

Disadvantages

- Longer compile times change developer experience from .NET

Ahead-of-time compiled generics

Bartok, like CLR, uses a non-uniform representation strategy

- Value types have many different sizes
- Specialize generic for each different layout
- Avoids unexpected boxing of values (heap allocation) in systems code

CLR lazily instantiates generic types

- Relies on JIT to build specialized generic code "as needed"
- NGEN can always fall back on this

Challenge for ahead-of-time compilation: instantiating all generic types at compile time

- Was not clear how well this would work: large number of generics possible
- Interacts with library story (shared code)

Example

```
public class HashMap<TKey, TValue> : Map<TKey,TValue>
{
    readonly HashingComparer<TKey> m_comparer;
    MapEntry<TKey, TValue>[] m_entries =
          EmptyArray<MapEntry<TKey, TValue>>.Instance;
    int m_count;
    int[] m_buckets;

    public override bool TryGet(TKey key, out TValue value)

    ....
}

struct MapEntry<TKey, Value>
{
      public int HashCode;
      public int NextEntry;
      public TKey Key;
      public TValue Value;
}
```

HashMap<string, int> creates these types:

Map<string, int>, MapEntry<string,int>,
HashingComparer<string>,
EmptyArray<MapEntry<string,int>>

MapEntry<string ,int> size 20 bytes (on x64)
- HashCode: offset 0, size 4
- NextEntry: offset 4, size 4
- Key: offset 8, size 8, GC pointer
- Value: offset 16, size 4

HashMap<string, string> creates these types:

Map<string,string>, MapEntry<string,string>,
HashingComparer<string>,
EmptyArray<MapEntry<string,string>>

MapEntry<string,string> size 24 bytes (on x64)
- HashCode: offset 0, size 4
- NextEntry: offset 4, size 4
- Key: offset 8, size 8, GC pointer
- Value: offset 16, size 8, GC pointer

Implementing generics

Prove that number of generic instantiations is finite

Each generic instantiation gets its own vtable

C# has runtime type identity

VTables can be costly, rivalling machine code in size in an executable

Bartok merges VTables when it can prove type identity does not matter

Structurally identical instantiations share code for methods (generic sharing).

Bartok uses type passing (dictionaries) like CLR does for runtime operations dependent on type (e.g. new(T)).

Sharing decision made on a per-method basis.

Share code for instantiations with value type args, in addition to reference type args

Value types popular in Midori systems programming (no heap allocation)

Same size, alignment, GC pointers, and calling convention.

Empirical evaluation of generics

Compare native executable sizes to input MSIL sizes

MSIL has one copy of the code for a generic type

Native code has specialized (instantiated) copies

Evaluate ratio for 3 Midori builds

Generics are used: generic methods are about 30% of native code size

	Native/MSIL ratio	Native (bytes)	MSIL (bytes)
Release-x64	3.24	197,595,648	60,954,624
Release-x86	2.73	165,058,048	60,428,288
Release-Tegra3	2.33	111,294,976	47,695,872

Approaches

Compile ahead-of-time

Reduce the memory and time overheads of managed code

Top 5 features:

- Generic sharing
- **Shared libraries**
- **Class initialization at process start-up**
- **Frozen objects**
- **Efficient linked stacks for concurrency**

Will discuss shared library implementation in more depth

Highly optimizing compiler

- Same optimization capabilities as production C++ compilers.
- Extend for managed code.

Shared libraries

Programmers expect to have large, rich class libraries

How do we have lots of small processes then?

Midori new process memory footprint: 145 Kbytes

Solution: have libraries that are shared across *many* processes

Code is loaded once and used many times, which amortizes memory footprint of code

To provide the rich experience, allow any OO type to be exported

Including generics, which complicate things.

Windows exports flat interfaces

Midori trade-off: library changes may be breaking changes at the binary level.

App stores make this reasonable to consider

Can recompile apps if library changes

Midori shared libraries

Currently 21 shared libraries. Some examples for x64:

Name	Description	MSI size	Native size
PlatformCore	Core functionality	8,479,000	11,228,160
PlatformGraphics	Graphics stack	6,343,168	4,293,120
PlatformNET	Networking	602,624	2,306,650
PlatformStorage	File system	1,239,040	5,480,960
PlatformWebRuntime	HTML rendering	3,423,232	8,045,568

Compilation model

Applications/libraries are brittle with respect to libraries they use

- Application/library may need a new generic instantiation for a value type.
- Application/library may subclass a type from a library
- If a library changes, its consumers must be recompiled.

This cuts both ways, though:

We use information about libraries during optimization.

- Cross compilation unit inlining of method bodies
- Existing generic methods

Class initialization at process start-up

C# class constructor semantics lead to runtime checks

Lazy class initialization has checks before field accesses and static/instance method calls.

"Before field init" has checks before field accesses

These checks don't exist for C++ and lead to "peanut butter costs"

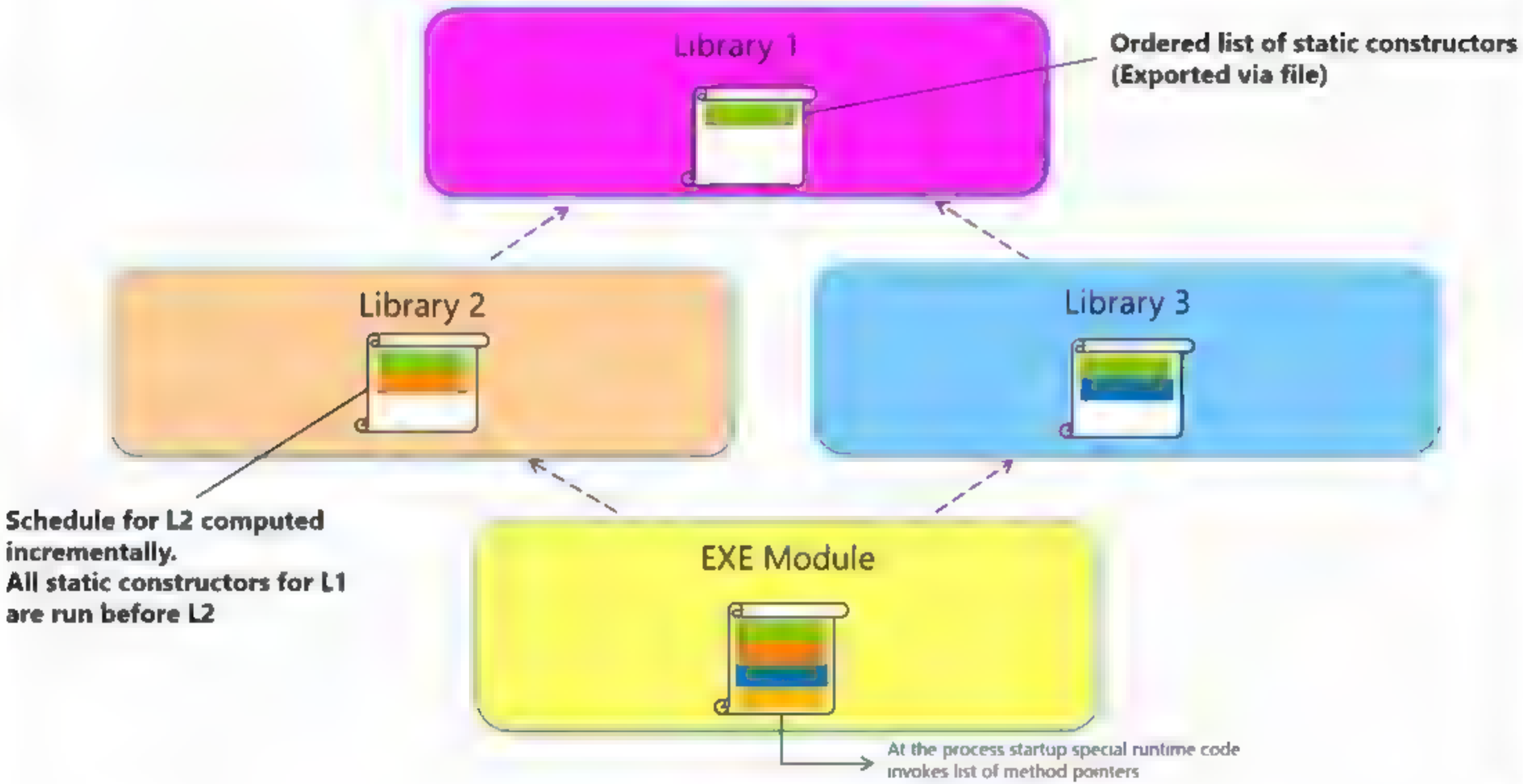
To avoid these checks, Midori initializes all classes at process start up.

Static constructors have no capabilities in Midori, so they can't do I/O

Compiler analysis decides on class initialization order:

- Build dependency graph for class constructor code
- Do topological sort, reject compilation units with cycles.
- Worst-case assumptions about methods that "escape" to prior compilation unit during class construction.

Static Constructor Scheduling and Shared Libraries



Frozen objects

Statically-initialized readonly data is valuable

- Improve process start up times: avoid cost of running class constructors
- Reduce memory footprint: share data across all processes using library
- GC friendly

Try to evaluate each class constructor at compile time

Results in:

- Initialized static fields pointing to a graph of objects in read-only memory (or containing immutable value types)
- Or a decision to defer initialization of that constructor to process start up.

Optimization depends on Midori's TSE type system

Allows programmers to specific immutability of object graphs and fields

Compile-time code execution

Only done when optimizations are enabled

Interprets code of static constructors at compile time

- It doesn't process MSIL: Bartok IR is used.
- Platform independent evaluation
- Interpreter is comprehensive: only native code calls, exceptions, some unsafe code operations aren't supported

Compiler loads types (including code) from libraries that are reachable from class constructors

Done after creation of generic instantiations and shared methods

Efficient linked stacks

Midori can have thousands of async computations in flight at once.

- Can't have a large stack per async computation.
- Solution is to break stacks into segments and link them together dynamically [Von Behren, SOSP '03]
- Explicit stack checks and linking can cost 2-5% of performance (more peanut butter)

Improve linked stacks using asymmetry between async vs. sync methods

Type system differentiates between the two kinds of methods

Synchronous methods can never block

Execute async methods on linked stacks

For synchronous methods, switch to a large stack with a guard page

No checks needed!

Linked stack optimization

Modified calling convention

- Caller must provide a minimum amount of stack space
- Optimizes small functions

Minimize stack check placements

- Push checks up call graph using interprocedural analysis to calculate max stack required
- Propagate requirements across libraries

Optimize switching between stacks

- Stack switch injection driven by register allocator
- Treat stack pointer as an implicit call argument that has one of two possible values
- Register allocator minimizes *dynamic cost of reloading values to registers*,
  - It effic ently places the reload (switch)
- Important runtime methods are stack neutral, no switch necessary

Approaches

Compile ahead-of-time

Reduce the memory and time overheads of managed code

Top 5 features:

- Generic sharing
- Shared libraries
- Class initialization at process start-up
- Frozen objects
- Efficient linked stacks for concurrency

Will discuss shared library implementation in depth

Highly optimizing compiler

- Same optimization capabilities as production C++ compilers.
- Extend for managed code.

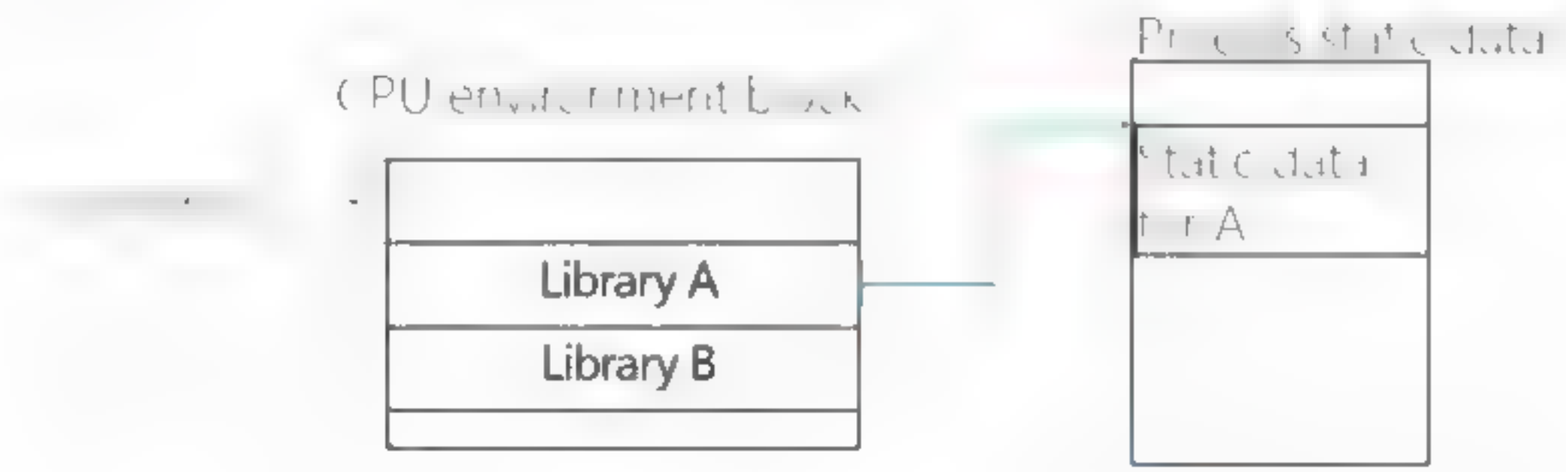
Implementing shared libraries

A library is a set of MSIL assemblies compiled into one native image

Defines methods, static fields, read only data, and runtime metadata for types

Static fields:

- Need per-process instance of field.
- Midori processes are in a *single* address space.
- Implement in software using a CPU environment block



CPU Environment Block

Example code:

```
// Reading a field in non-generic class D in A
base_A = FS:[s_A]
value = [base_A + field_offset]
```

Slot numbers need to be global across domain

A library needs to finds its static data at the same slot in every process

Slot numbering isn't known at compile time

Number of libraries on deployed system isn't known.

Would have to waste per-process space on libraries potentially not in use.

Loader numbers libraries, embeds numbers into code during loading.

Accessing code and read-only data in library

Everything is in same address space

- Suppose executable E uses library L
- Loader fixes up E to point directly to the code/data for L in memory

Libraries and generics

Who defines an instantiation used in multiple libraries?

Design choice: self-contained libraries and executables

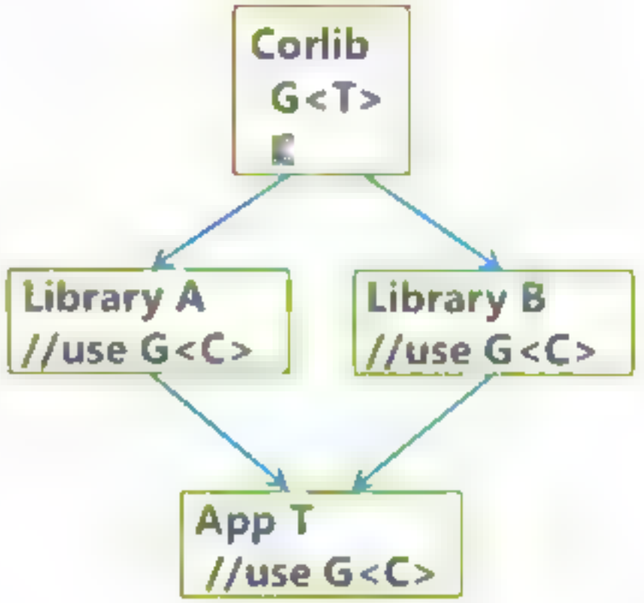
New instantiations always get code and vtables.

Multiple copies of code, vtables can exist at runtime

Re use instantiations from prior compilation units

Simplifies system

- No system wide information / tables
- Can always unload unused libraries



Identify potentially multiply instantiated (PMI) types

"Pay as you go": only they pay when possible

Instantiation is not a PMI if any type syntactically occurring in it is defined in current compilation unit

PMI implementation

Type tests

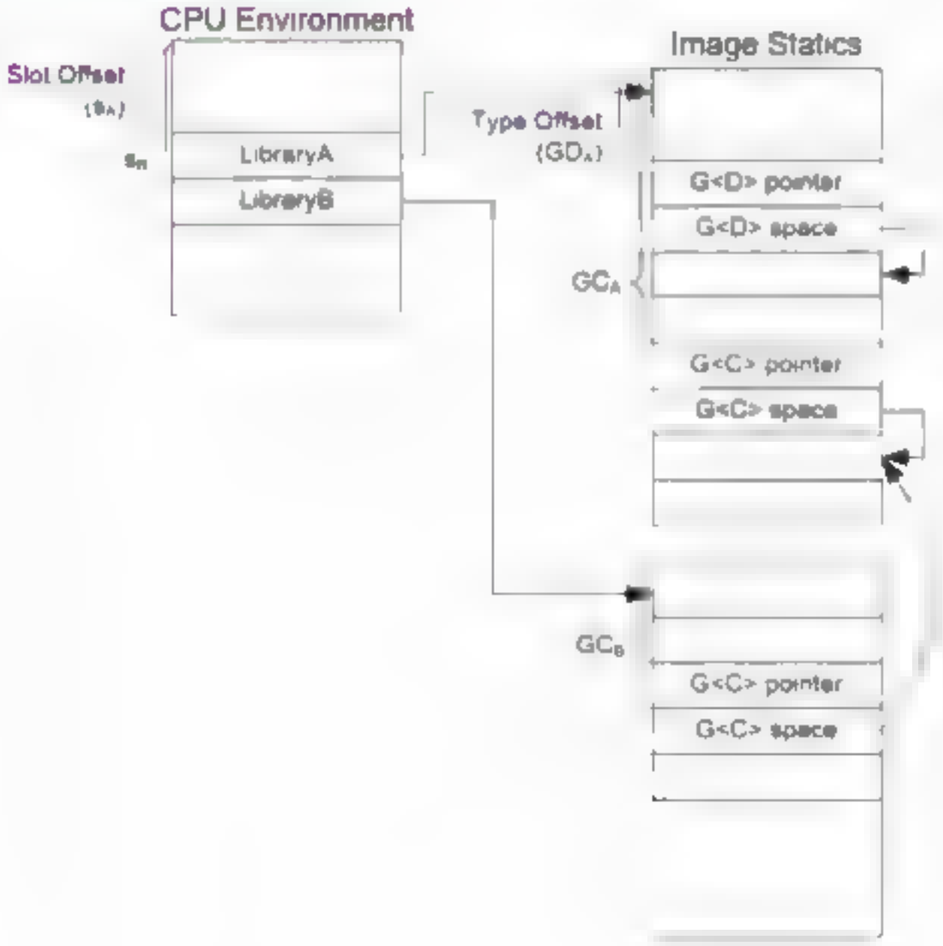
- Use vtable pointer equality first.
- If that fails, use expensive structural type test
  - Use per type hashcode to speed up negative structural type tests
- Have to check for PMIs along some fast paths

Static fields

- Add another level of indirection to PMI field access
- When compiling an executable, compiler:
  - For each PMI instantiation, picks one occurrence as the representative instantiation.
  - Sets up per library tables of representative instantiations
  - Used at class initialization to initialize the extra indirection.

PMI static fields

Here G<C> is a PMI:



// Accessing a multiply-instanced field in G<C>

$base_A = FS:[s_A]$

$indirect_{GC} = [base_A + GC_A + pointer\_offset]$

$value = [indirect_{GC}]$

Approaches

Compile ahead-of-time

Reduce the memory and time overheads of managed code

Will discuss top five features we added for this:

- Generic sharing
- Shared libraries
- Class initialization at process start-up
- Frozen objects
- Efficient linked stacks for concurrency

Will discuss shared library implementation in depth

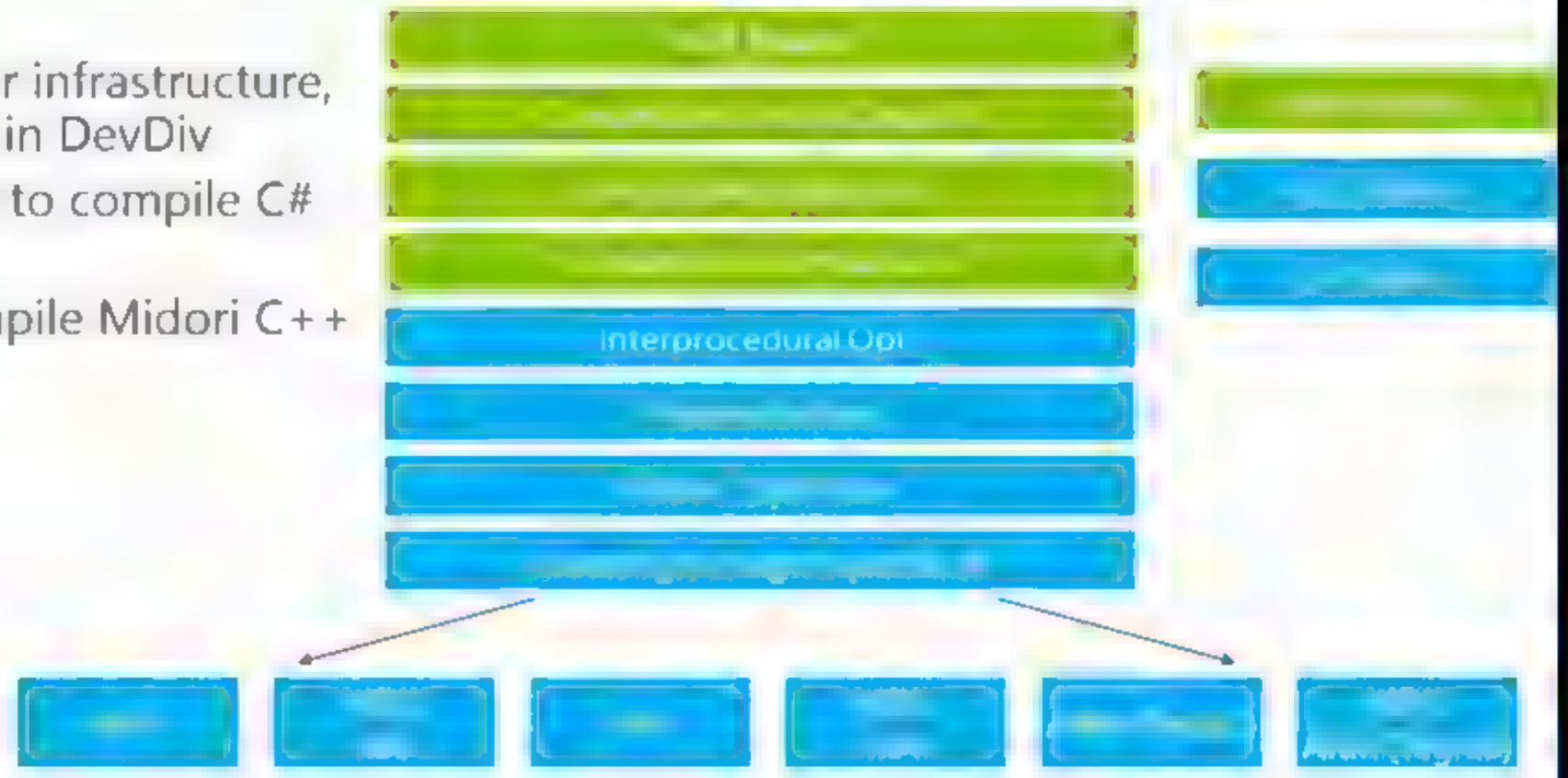
Highly optimizing compiler

- Same optimization capabilities as production C++ compilers.
- Extend for managed code.

Optimizing compiler

Bartok is the fusion of two optimizing compilers

- Original MSR compiler, used by Singularity
- The Phoenix compiler infrastructure, originally developed in DevDiv
- Phoenix can be used to compile C# and C++ code
- Phoenix used to compile Midori C++ code.



Optimizing compiler

Modes

- Library
  - Assume everything public or protected is called
- Executable
  - Assume only entry points are called externally
  - May use information from library compilation
- Whole program
  - Assume only entry points are called
  - Assume all code is known

Profile-guided optimization

- Orthogonal to the mode
- Done only for Phoenix phases
- All phases are profile aware

Optimizing compiler

Bartok is the fusion of two optimizing compilers

- Original MSR compiler, used by Singularity
- The Phoenix compiler infrastructure, originally developed in DevDiv
- Phoenix can be used to compile C# and C++ code
- Phoenix used to compile Midori C++ code.

Optimizing compiler

Modes

- Library
  - Assume everything public or protected is called
- Executable
  - Assume only entry points are called externally
  - May use information from library compilation
- Whole program
  - Assume only entry points are called
  - Assume all code is known

Profile-guided optimization

- Orthogonal to the mode
- Done only for Phoenix phases
- All phases are profile aware

Optimizations

Intraprocedural

- Expression simplification
- Expression reshaping
- Copy/constant propagation
- Constant folding
- Unreachable code
- Dead code elimination
- CSE
- Value numbering
- Partial redundancy elimination (PRE)
- Array bounds check elimination
- Control flow opts
- Type test elimination
- Range analysis (for eliminating safety checks)

Machine-level

- Hierarchical tiling-based register allocation
- Stack packing
- Instruction scheduling
- Machine idioms
- Code layout
- Optimized struct copying/initialization
- Optimized block copying/zeroing

Optimizations

Interprocedural

- Inlining
- Tree shaking (instantiation and invocation)
- Devirtualization using class/method hierarchy analysis
  - Hierarchy analyses extending to generics
- Stack allocation
- Constant propagation
- Null check elimination
- Interprocedural range analysis
- Return value optimization
- Bottom up summary information

Loop optimizations

- Loop invariant removal
- Strength reduction + induction variable elimination
- Loop unrolling
- Loop versioning

Managed-code specific

- Write barrier insertion + specialization
- Specialized type tests
- GC synchronization analysis
- Arithmetic check folding
- Runtime metadata elimination
- Runtime vtable merging

Image size

Midori x64 build

- Approximately 500 programs
- Includes:
  - Browser
  - Shell
  - SpecWeb
  - Tetris
  - Most tests

	Native/MSIL ratio	Native (bytes)	MSIL (bytes)
Release-x64	3.24	197,595,648	60,954,624
Release-x86	2.73	165,058,048	60,428,288
Release-Tegra3	2.33	111,294,976	47,695,872

Effect of generic sharing

Generic sharing over reference types: 23% reduction

Add generic sharing over value types: 10% reduction

Total reduction: 33%

(This is for an older version of the system)

Shared libraries

Before shared generics and shared libraries, Midori was>800 Mbytes (with much less functionality)

Reduction from sequentially adding libraries one at a time:

Base	48%
UserInterface	22%
Testing	16%
Cumulative	65%

(This is from an older version of the system.)

Midori is a feasible OS today because of shared libraries

C# Integer Benchstone x64

benchi /O2	JIT seconds (lower is better)	NGEN seconds (lower is better)	Feb-2013 seconds (lower is better)	JIT/Feb-2013 (higher is better)	NGEN/Feb-2013 (higher is better)
8queens.c	4.79	4.85	5	0.96	0.97
ackerman.c	5.33	4.65	4.53	1.18	1.03
addarra2.c	17.23	17.22	13.93	1.24	1.24
addarray.c	21.45	21.44	14.91	1.44	1.44
array1.c	8.42	8.43	7.86	1.07	1.07
array2.c	30.76	30.85	48.04	0.64	0.64
benche.c	15.6	14.79	10.64	1.47	1.39
binserch.c	10.71	10.01	7.68	1.39	1.30
bubsort.c	17.75	17.73	13.82	1.28	1.28
bubsort2.c	13.81	14.59	10.82	1.28	1.35
csieve.c	11.4	11.38	9.15	1.25	1.24
fib.c	9.34	7.81	0.01	934.00	781.00
heapsort.c	13.1	12.58	11.12	1.18	1.13
iniarray.c	27.52	27.02	2.66	10.35	10.16
logicarr.c	22.42	22.46	16.81	1.33	1.34
midpoint.c	18.46	18.49	21.02	0.88	0.88
mulmtx.c	20.65	20.66	13.42	1.54	1.54
ndhrysto.c	27.69	28.64	17.59	1.57	1.63
permutat.c	8.14	10.14	7.21	1.13	1.41
pi.c	10.95	10.95	8.6	1.27	1.27
puzzle.c	14.5	14.64	12.77	1.14	1.15
quicksrt.c	12.86	12.71	11.27	1.14	1.13
shellsrt.c	24.4	24.26	20.86	1.17	1.16
sq_mtx.c	17.67	17.75	9.47	1.87	1.87
treesort.c	3.83	3.69	3.76	1.02	0.98
tree_ins.c	9.17	9.16	9.29	0.99	0.99
xpos_mtx.c	34.04	34.01	30.13	1.13	1.13
"rw"-Geomean	**14.07**	**14.08**	**11.36**	**1.24**	**1.24**
Geomean	**13.94**	**13.83**	**8.40**	**1.66**	**1.65**

SPEC2K6 x64 /O2 vs. VS

CPU2006 INT /O2 i7	LKG3 seconds (lower is better)	Feb-2013 seconds (lower is better)	LKG3/Feb-2013 (higher is better)	LKG3 bytes (lower is better)	Feb-2013 bytes (lower is better)	Feb-2013/LKG3 (lower is better)
astar	605.5	595	1.02	93893	97440	1.04
bzip2	806.6	791.6	1.02	102222	100645	0.98
[illegible]	554.8	533.6	1.04	2350965	2313612	[illegible]
gobmk	777.2	723.7	1.07	677732	661880	[illegible]
h264ref	871.3	814.3	1.07	496964	514059	1.03
hmmer	484	[illegible]	0.79	204784	195730	0.96
libquantum	874.3	839.7	1.04	94143	92586	[illegible]
mcf	452.6	[illegible]	0.97	69112	69176	1.00
omnetpp	415.4	401.3	1.04	547996	554201	1.01
perlbench	718.1	[illegible]	1.05	924626	932910	1.01
sjeng	870.4	834.9	1.04	154147	149747	0.97
xalancbmk	326.4	301.3	1.08	2157540	2144857	0.99
Geomean	**615.9**	**[illegible]**	**1.02**	**330,628**	**[illegible]**	**1.00**

CPU2006 FP /O2 i7	LKG3 seconds (lower is better)	Feb-2013 seconds (lower is better)	LKG3/Feb-2013 (higher is better)	LKG3 bytes (lower is better)	Feb-2013 bytes (lower is better)	Feb-2013/LKG3 (lower is better)
dealII	515.4	522.3	0.99	573876	621763	1.08
lbm	449	453.9	0.99	87874	89282	1.02
milc	615.9	618.7	1.00	155592	164001	1.05
namd	720.2	705	1.02	375944	362612	0.96
[illegible]	392.8	332.6	1.18	934922	919081	0.98
soplex	379.8	368	1.03	385832	374103	0.97
sphinx3	893.5	878.3	1.02	203693	197933	0.97
Geomean	**541.7**	**525.8**	**1.03**	**299,614**	**301,156**	**1.01**

Conclusions

Bartok is the compiler underlying Midori:

Ahead-of-time compilation of C#

Including generics, which are fully instantiated at compile time

Features that reduce the memory and time overhead of managed code

Shared libraries are required for Midori to be a viable OS
If you have generics, you must have generic sharing
Avoid peanut butter

Highly optimizing compiler

Further reading

http://midori/Midori%20Design%20Notes/Forms/AllItems.aspx:

MDN 95: Midori Code Sharing and Separate Compilation Model

MDN 143: Bartok Generics

MDN 225: Frozen Objects

MDN 226: Polymorphic Hierarchy

MDN 228: Implementing the Midori Asynchronous Programming Model